THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Dispel4Py: A Python Framework for Data-intensive eScience

Citation for published version:
Krause, A, Filgueira, R & Atkinson, M 2015, Dispel4Py: A Python Framework for Data-intensive eScience. in *Proceedings of the 5th Workshop on Python for High-Performance and Scientific Computing.*, 6, ACM, New York, NY, USA. https://doi.org/10.1145/2835857.2835863

Digital Object Identifier (DOI):
10.1145/2835857.2835863

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the 5th Workshop on Python for High-Performance and Scientific Computing

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 05. May. 2023

dispel4py: A Python Framework for Data-Intensive eScience

Amrey Krause
EPCC
University of Edinburgh
Edinburgh EH9 3FD, UK
a.krause@epcc.ed.ac.uk

Rosa Filgueira
School of Informatics
University of Edinburgh
Edinburgh EH8 9LE, UK
rosa.filgueira@ed.ac.uk

Malcolm Atkinson
School of Informatics
University of Edinburgh
Edinburgh EH8 9LE, UK
mpa@staffmail.ed.ac.uk

ABSTRACT

We present `dispel4py`, a novel data intensive and high performance computing middleware provided as a standard Python library for describing stream-based workflows. It allows its users to develop their scientific applications locally and then run them on a wide range of HPC-infrastructures without any changes to the code. Moreover, it provides automated and efficient parallel mappings to MPI, multiprocessing, Storm and Spark frameworks, commonly used in big data applications. It builds on the wide availability of Python in many environments and only requires familiarity with basic Python syntax. We will show the `dispel4py` advantages by walking through an example. We will conclude demonstrating how `dispel4py` can be employed as an easy-to-use tool for designing scientific applications using real-world scenarios.

1. INTRODUCTION

Recent years have seen a spectacular growth in scientific data, that must be shared, processed and managed on different distributed computational infrastructures (DCI). Major contributors to this phenomenal data deluge include new avenues of research and experiments facilitated by e-Science. Such e-Science data emanates from different areas, such as sensors, satellites, high-performance computer simulations and already exceeds tens of petabytes per year. However, success with these technologies depends on additional mechanisms that are not straightforward for non-experts: for example MPI or OpenMP) should be used depending on the memory architecture. This technical detail distracts from the domain goals and limits progress, *e.g.* by requiring code changes for each target DCI.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
PyHPC2015, November 15-20, 2015, Austin, TX, USA
©2015 ACM.
ISBN 978-1-4503-4010-6/15/11...$15.00
DOI: http://dx.doi.org/10.1145/2835857.2835863.

This paper presents a new toolkit for scientists, called `dispel4py`, to enable them to rapidly prototype their distributed data-intensive applications. It provides an enactment engine that maps and deploys abstract workflows onto multiple parallel platforms, including Apache Storm, MPI and shared-memory multi-core architectures.

`dispel4py` has been primarily used in e-Science contexts, most notably in *Seismology*.

This paper is structured as follows. Section 2 presents the motivation for using the Python language. Section 3 presents the VERCE project. Section 4 defines the `dispel4py` concepts and design. Section 5 shows how to install and use `dispel4py` walking through an example workflow. Section 6 discusses `dispel4py` mappings. Section 7 presents two `dispel4py` workflows in the Seismology domain. Section 8 introduces the `dispel4py` monitoring service. We conclude with a summary of achievements and outline some future work.

2. WHY PYTHON?

Python is a high-level programming language, interpreted with capabilities for object-oriented programming. Python has a simple, easy to learn syntax, emphasises readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. Often scientists choose this language because of the simplicity to quickly code their scripts hence providing increased productivity. Since there is no compilation step, the edit-test-debug cycle is incredibly fast.

In particular, Python is very popular in the e-Science domain of Seismology. Moreover, Python provides a plethora of rich scientific computing libraries such as:

- NumPy[1]: Python library for array processing for numbers, strings, records, and objects
- SciPy[2]: Python library of algorithms and mathematical tools
- ObsPy[3]: A Python framework for processing seismological data.

There are also libraries for multitasking and parallel processing such as multiprocessing and MPI. However, these

[1]http://www.numpy.org/
[2]http://www.scipy.org/
[3]https://github.com/obspy/obspy/wiki

operate at a low level, depend on knowledge on the machine architecture and require the user to be familiar with the mechanisms. Our objective is to help scientists to write and put together higher level functionality, using their preferred environment for Python development, and testing it first on their own laptop, then moving on to the department cluster or an HPC service.

3. VERCE PROJECT

The EU VERCE project[4], "Virtual Earthquake and seismology Research Community in Europe e-science environment", has developed a comprehensive and integrated virtual research environment (VRE) for computational and data-intensive seismology. This has been pioneered with particular simulation models and data-driven seismology examples.

The present VERCE e-Infrastructure involves major elements moving from the researchers' point of contact to the contextual digital resources.

The scientific gateway that is web accessible from anywhere, provides an integrated view of all available resources, handles continuity between sessions and supports collaboration with shared data and methods, and with pervasive data access controls. For scalable data processing it leverages the data-intensive Python framework `dispel4py` described in this paper. `dispel4py` enables mapping to multiple Distributed Computing Infrastructures (DCIs) within the infrastructure.

4. DISPEL4PY CONCEPTS

It is increasingly apparent that workflow tools and languages play significant roles due to their inherent modularity and intuitive visualisation properties – graphs appear to be a natural way to visualise logical compositions of processing steps. We are just experiencing a critical transition from an era when performance issues dominated to an era when these can be dealt with automatically so that domain scientists can experiment with and take full responsibility for the encoding of their methods. We adopt the abstract model of data streaming between operations as it is versatile and easily understood. It will have a significant impact on the way scientists think and carry out their data-analysis tasks. The low overhead of the interconnections make it suitable for immediate interpretation and for composing small as well as large steps, yet data streams can be expanded to arbitrary capacity and the graphs are easily parallelised. Consequently, `dispel4py` allows scientists to express their requirements in abstractions closer to their needs and further from implementation and infrastructural details.

The Dispel language [2] had these goals. `dispel4py` builds on this but aligns more closely with the requirements of scientists by providing them with a toolkit in Python, a familiar programming language, to extend their existing data processing scripts into streaming workflows easily and intuitively.

In the following we present a summary of the main `dispel4py` concepts and terminology. More information is available at the `dispel4py` documentation[5].

4.1 Processing Elements

A processing element (PE) is a computational activity, corresponding to a step in a scientific method or a data transforming operator. A PE encapsulates an algorithm or a service. PEs are the basic computational blocks of any `dispel4py` workflow at an abstract level and form the nodes in a workflow graph. `dispel4py` offers a variety of base classes for PEs to be extended: GenericPE , IterativePE, ConsumerPE, SimpleFunctionPE and CompositePE. The primary differences between them are the number of inputs that they accept and how users express their computational activities.

GenericPE represents the basic interface and accepts a configurable number of inputs and outputs, whereas IterativePE declares exactly one input and one output and usually encapsulates a simple iterative computational step such as a filter or a transformation. A ConsumerPE has one input and no outputs, representing a leaf in the workflow tree and accordingly, a ProducerPE has no inputs and one output and serves as a data producer or root. When extending any of these PE classes the user overrides the _process method. The SimpleFunctionPE behaves like an IterativePE and wraps a function, providing an even easier way to create iterative computational building blocks by simply implementing a function (rather than creating a PE class). A CompositePE contains a subgraph and declares inputs and outputs in the same way as any other PE.

4.2 Instances

An instance is the executable copy of a PE that will consume data units from its input ports and emit data units from its output ports transformed by its algorithm. During enactment and prior to execution each PE is translated into one or more instances. Multiple instances may occur to parallelise execution and increase data throughput.

4.3 Connection

A connection streams data units from an output port of a PE instance to one or more input ports of other instances. The rate of data consumption and production depends on the behaviour of the source and destination PEs. A PE must declare its connections that it provides as this is required information when the graph is translated into the enactment process.

4.4 Composite PEs

A composite processing element is a PE that wraps a `dispel4py` sub-workflow. Composite processing elements allow for synthesis of increasingly complex workflows by composing previously defined sub-workflows. They present as PEs and may be used to hide complexity from the user.

4.5 Workflow graph

A graph defines the ways in which PEs are connected and hence the paths taken by data, *i.e.* the topology of the workflow. There are no limitations on the type of graphs that can be designed with `dispel4py`. Figure 2 is an example graph with four PEs in a simple pipeline. The "root" PE readRaDec, or data producer, starts the pipeline by reading input parameters and passes them to getVOTable as inputs. This PE emits the galaxy data as a VO table whose columns are then filtered by filterColumns and finally the internal extinction of the galaxy is computed. We explain this example in more detail in section 5. More examples can be found in

[4]EU VERCE, http://www.verce-project.eu, RI 283543.
[5]http://dispel4py.org

the dispel4py documentation[6].

4.6 Grouping

A grouping specifies the communication pattern between connected PEs and specifies how the data stream is distributed to a number of parallel instances. These patterns are relevant at the parallel enactment stage as each pattern arranges the set of receiving PE instances. There are four patterns available: shuffle, group_by, one_to_all and all_to_one.

- The shuffle grouping randomly distributes data units to the receiving instances and, depending on the enactment platform, this may be optimised according to load.
- The group_by grouping ensures that data units with certain features or values are received by the same instance, thus enabling aggregations similarly to the relational algebra context.
- In the one_to_all grouping all PE instances receive copies of all output data from the connected instances. This is equivalent to a broadcast in the MPI context.
- all_to_one means that all data units are received by a single instance, for example to provide a total sum in an aggregation.

Consider the classic WordCount workflow which counts the words in a document. In the dispel4py workflow the WordCount PE applies a group_by grouping based upon the field word to collect the number of occurrences of a word, and the final PE applies an all_to_one grouping to collect the output.

4.7 Partitions

A partition is a number of PEs wrapped together and is usually executed within the same process. It may be used to explicitly co-locate PEs that have relatively low CPU and RAM demands, but high data-flows between them. This corresponds to task-clustering in the task-oriented workflow systems. Note that *partitions* provide an optimisation mechanism during enactment whereas *composite PEs* represent reusable components at a logical level.

5. USING DISPEL4PY

This section describes details about the software package dispel4py, how to install it, where to find documentation, and we walk through an example script that constructs a workflow.

5.1 Installation

The latest stable release of dispel4py (currently version 1.2) is available from the Python Package Index (PyPI)[7]. dispel4py is pure Python and supports Python versions 2 and 3 and can be executed in any environment where Python is available, notably recent Linux versions on which most cloud clusters are based, and HPC services.

dispel4py is installed via the Python package management system pip using a simple command:

```
pip install dispel4py
```

The documentation is available on the dispel4py website[8].

dispel4py project files structure

```
[dispel4py]
├── __init__.py
├── __main__.py
├── base.py
├── core.py
├── new
│   ├── __init__.py
│   ├── aggregate.py
│   ├── mappings.py
│   ├── monitoring.py
│   ├── mpi_process.py
│   ├── multi_process.py
│   ├── processor.py
│   ├── simple_process.py
│   └── spark_process.py
└── storm
    ├── __init__.py
    ├── client.py
    ├── storm_submission.py
    ├── storm_submission_client.py
    ├── topology.py
    └── utils.py
```

Figure 1: dispel4py project files structure

5.2 Modules

The package contents of dispel4py are shown in Figure 1.

- The core module defines the base class GenericPE which all PE implementations must extend.
- The base module makes available the utility PE classes described in the previous section, notably IterativePE, ConsumerPE, CompositePE.
- The module workflow_graph constructs the graph representation, building on the Python package networkx[9]. It also provides other utilities, for example visualisation of the dispel4py graph using Graphviz dot[10].
- The mapping modules mpi_process (MPI), multi_process (shared memory multi-process) and simple_process for sequential mapping are described in the following section. They are based on the shared processor module containing methods applicable to all platforms.

[6] http://dispel4py.org/documentation/dispel4py.examples.graph_testing.html
[7] https://pypi.python.org/pypi/dispel4py
[8] http://dispel4py.org/documentation/
[9] https://networkx.github.io/
[10] http://www.graphviz.org/

- The Storm mapping has its own package with various modules that translate a `dispel4py` workflow to a Storm topology and submit it to a cluster.

5.3 Building workflows

Users construct their workflows as standalone Python scripts that describe the graph and import dependencies as required. Note that the dependencies must be available on the target enactment system but they may not be supported by the client system, enabling users to submit workflows to a different system than their own.

When creating a `dispel4py` workflow graph, it is necessary to first instantiate a `WorkflowGraph` object and then add PEs to it by connecting them together (see example code listing below). Users may use available PEs from the `dispel4py` libraries, or implement their own PEs (in Python) if they require new functionality.

In the interactive IPython[11] shell `dispel4py` supports the visualisation of a constructed workflow graph using Graphviz dot that can be inspected by the user for correctness.

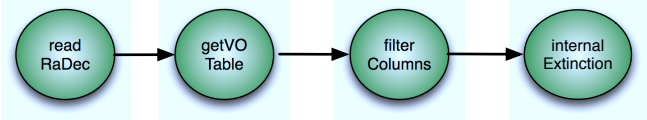


Figure 2: Internal Extinction `dispel4py` workflow.

As an example, we show the code for constructing the workflow illustrated in Figure 2. This is a workflow in the astronomy domain that queries a Virtual Observatory (VO) service[12] for a number of galaxies and calculates their *internal extinction*. The PEs are connected in a pipeline. In the code listing shown we assume that the logic within PEs is already implemented.

```
from dispel4py.workflow_graph \
import WorkflowGraph

pe1 = ReadRaDec()
pe2 = GetVOTable()
pe3 = FilterColumns()
pe4 = InternalExtinction()

graph = WorkflowGraph()
graph.connect(pe1,'data',pe2,'input')
graph.connect(pe2,'votable',pe3,'votable')
graph.connect(pe3,'columns',pe4,'input')
```

Once the `dispel4py` workflow has been built, it can be automatically executed in several distributed computing infrastructures leveraging the mappings that are explained in the next section.

6. DISPEL4PY FEATURES

We will describe `dispel4py`'s *mapping* features that allow users to automatically run their workflows with different enactment systems.

One of `dispel4py`'s strengths is that it allows the construction of workflows without knowledge of the hardware or middleware context in which they will be executed. Users focus on designing their workflows at a logical level, describing actions, input and output streams, and how they are connected. When their Python script is run this graph is constructed, and then it is either locally interpreted or the `dispel4py` system maps the graph to a selected enactment platform. Since the abstract workflows are independent from the underlying communication mechanism they are portable among different computing resources without any migration cost imposed on users, i.e. they do not make any changes to run in a different context.

The `dispel4py` system currently implements mappings for MPI, shared memory DCIs, Apache Storm and a prototype Apache Spark, as well as a Sequential mapping for development and small applications.

6.1 MPI

MPI is a standard, portable message-passing system for parallel programming, whose goals are high performance, scalability and portably [4]. For this mapping, `dispel4py` uses mpi4py[13], which is a full-featured Python binding for MPI based on the MPI-2 standard. The `dispel4py` system maps PEs to a collection of MPI processes. Depending on the number of targeted processes, which the user specifies when executing the mapping, multiple instances of each PE are created to make use of all available processes. Input PEs, i.e. at the root of the `dispel4py` graph, only ever execute in one instance to avoid the generation of duplicate data blocks.

Data units to be shipped along streams are converted into pickle-based Python objects and transferred using MPI asynchronous calls. Groupings are mapped to communication patterns, which assign the destination of a stream (*e.g.* shuffle grouping is mapped to a round-robin pattern, for group-by the hash of the data block determines the destination). For both MPI and multiprocessing, users can specify partitions of the graph and the mapping distributes these across processes in the same way as single PEs. The MPI mapping requires mpi4py and any MPI interface, such as mpich[14] or openmpi[15].

6.2 Multiprocessing

The Python library multiprocessing is a package that supports spawning subprocesses to leverage multicore shared-memory resources. It is available as part of standard Python distributions on many platforms without further dependencies, and hence is ideal for small jobs on desktop machines, taking full advantage of multiple cores. The Multiprocessing mapping of `dispel4py` (also called multi) creates a pool of processes and assigns each PE instance to its own process. Messages are passed between PEs using multiprocessing.Queue objects. As in the MPI mapping, `dispel4py` maps PEs to a collection of processes. Each PE instance reads from its own private input queues on which its input blocks arrive. Each data block triggers the execution of the process() method which may or may not produce output blocks. Output from a PE is distributed to the connected PEs depending on the grouping pattern that the destination PE requests. The Communication class manages distribution

[11] http://ipython.org/
[12] http://www.ivoa.net/
[13] http://mpi4py.scipy.org/
[14] http://www.mpich.org/
[15] `http://www.open-mpi.org/`

of data. The default is ShuffleCommunication which implements a round-robin pattern; the GroupByCommunication groups output by specified attributes. The Multiprocessing mapping allows partitioning of the graph to colocate PEs in one process.

6.3 Apache Storm

The `dispel4py` system maps to Storm by translating its graph description to a Storm topology. As `dispel4py` allows its users to define data types for each PE in a workflow graph, types are deduced and propagated from the data sources throughout the graph when the topology is created. Each Python PE is mapped to either a Storm bolt or spout, depending on whether the PE has inputs (a bolt), i.e. is an internal stage, or is a data source (a spout), i.e. is a point where data flows into the graph from external sources. The data streams in the `dispel4py` graph are mapped to Storm streams. The `dispel4py` PEs may declare how a data stream is partitioned across processing instances. By default these instructions map directly to built-in Storm stream groupings. The source code of all mappings can be found at[16].

There are two execution modes for Storm: a topology can be executed locally using a multi-threaded framework (development and testing), or it can be submitted to a production cluster. The user chooses the mode when executing a `dispel4py` graph in Storm. Both modes require the Storm package on the client machine.

6.4 Apache Spark

Apache Spark[17] is a popular platform that leverages Hadoop YARN and HDFS taking advantage of many properties such as dynamic scaling and fault tolerance. It has also been used HPC platforms by distributing Spark worker nodes at runtime to the available processors of a job in an HPC cluster and managing Spark tasks. The Spark mapping of `dispel4py` is a prototype and it is targeted at users that are not familiar with the Hadoop/MapReduce environment but would like to take advantage of the rich libraries that the platform provides. The `dispel4py` system maps to Spark by translating a graph description to PySpark actions and transformations on Spark's resilient distributed datasets (RDDs). RDDs can be created from any storage source supported by Hadoop, such as text files in HDFS, HBase tables, or Hadoop sequence files. Root PEs in the `dispel4py` graph are mapped to RDD creators, and each PE with inputs is mapped to an action or a transformation of a RDD. At the leaves of the `dispel4py` graph a call to foreach() is inserted in order to trigger the execution of a complete pipeline of actions. In the future we envisage mapping a set of reserved PE names (possibly supported by the registry) to available actions and transformations in Spark to take full advantage of the optimisations available on the platform.

6.5 Sequential mode

The sequential mode (simple) is a standalone mode that is ideal for testing workflows during development. In sequential mode a `dispel4py` graph is enacted in sequence, respecting dependencies, within a single process without optimisation. When executing a `dispel4py` graph in sequential mode, the dependencies of each PE are determined and the PEs in the graph are executed in a depth-first fashion starting from the roots of the graph (data sources). The source PEs process a user-specified number of iterations or are supplied with an input dataset. All data is processed, and messages are passed between nodes, in-memory.

[16]https://github.com/dispel4py/dispel4py/

[17]http://spark.apache.org/

7. DISPEL4PY IN ACTION

The following subsections describe a `dispel4py` workflow from the Seismology domain to show how `dispel4py` enables scientists to describe data-intensive applications using a familiar notation, and to execute them in a scalable manner on a variety of platforms without modifying their code.

7.1 Seismic Noise Cross Correlation

Earthquakes and volcanic eruptions are sometimes preceded or accompanied by changes in the geophysical properties of the Earth, such as wave velocities or event rates. The development of reliable risk assessment methods for these hazards requires real-time analysis of seismic data and truly prospective forecasting and testing to reduce bias. However, potential techniques, including seismic interferometry and earthquake "repeater" analysis, require a large number of waveform cross-correlations, which is computationally intensive, and is particularly challenging in real-time.

With `dispel4py` we have developed the *Seismic Ambient Noise Cross-Correlation* workflow (also called the xcorr workflow) as part of the VERCE project [1], which preprocesses and cross-correlates traces from several stations in real-time. The xcorr workflow consists of two main phases:

- *Phase 1 – Preprocess*: A series of functions is applied to each continuous time series from a given seismic station (called a *trace*). The processing of each trace is independent from other traces, making this phase "trivially" parallel (complexity $O(n)$, where n is the number of stations).
- *Phase 2 – Cross-Correlation*: Pairs all of the stations and calculates the cross-correlation for each pair (complexity $O(n^2)$).

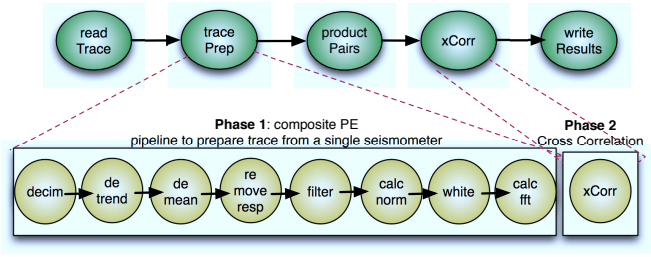


Figure 3: Cross Correlation `dispel4py` workflow.

Figure 3 shows the `dispel4py` xcorr workflow, which has five PEs. Note that the tracePrep PE is a composite PE, where data preparation (preprocessing) takes place. Each of those PEs within the composite tracePrep, from decim to calc_fft, performs processing on the data stream. The xcorr workflow was initially tested on a local machine using a small number of stations as input data. Later, it was executed and evaluated on different parallel platforms (described at Section 7.3 and summarised in Table 2) automatically, scaling up by using using the parallel mappings of `dispel4py`

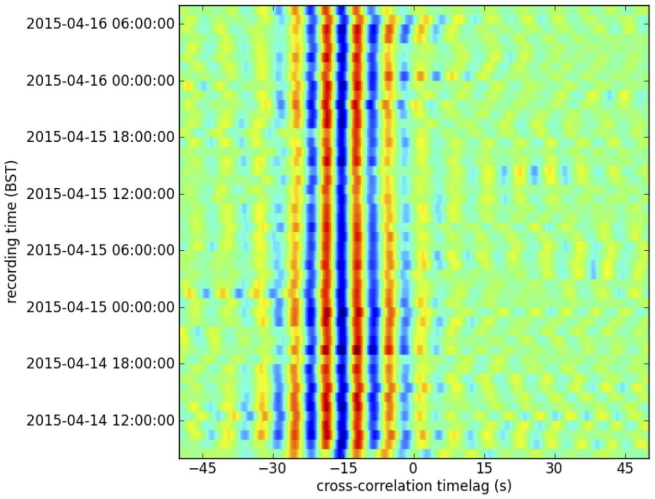


Figure 4: Cross Correlation `dispel4py` workflow.

to 1000 stations as input data (150 MB) performing 499,500 cross-correlations (39GB) without modifying the code.

As an exercise to test and validate the cross-correlation results, we chose two stations, which were live and sufficiently close together to each other to yield a coherent arrival package using just an hour of data, during three subsequent days (from 14/04/2015 to 16/04/2015). During the experiment, every hour the cross-correlation workflow was performed:

- downloading data from both stations,
- preprocessing the data,
- and finally cross-correlating them.

The results were gathered and visualized as shown in figure 4. The station pair was located in Southern California (one hugging the beachfront near LA and the other on an Island of the coast of LA). The results of this experiment shows that the noise is very directional, and we only create a one-sided (predominately surface wave) Green's Function Estimate.

Table 1: Measures (seconds) for 1000 stations on four DCIs with the maximum number of cores available

Mode	Terracorrelator	Amazon EC2	EDIM1	SuperMUC
MPI	3066.22	16862.73	38656.94	1093.16
multi	3143.77			
Storm		27898.89	120077.123	

The results (see Table 1) demonstrate that `dispel4py` can be applied to diverse DCI targets and adapt to variations among them. However, the xcorr performance depends on the DCI selected. For example, the best results in terms of performance were achieved on the SuperMUC machine with MPI followed by the Terracorrelator machine with MPI and multi mappings. The Storm mapping proved to be the least suitable in this case. Yet it is the best mapping in terms of fault-tolerance for any case and DCI, as Storm delivers automatic fault recovery and reliability. It may be those features that make it the slowest mapping. See [3] for further measurements.

7.2 Misfit Calculation

As in many sciences, seismologists observe data, infer possible physical models at the origin of the data and then compare the results of the modelling with the observations. For each event in the region, the propagation of seismic waves is simulated generating synthetic seismograms for each seismometer. The observed and synthetic traces are then compared in a process called *misfit analysis*. The detected differences can then be back-propagated to refine the model in a process called inversion. This can be done for all events in the region, and as the model of the Earth's structure improves it can be extended to finer resolution.

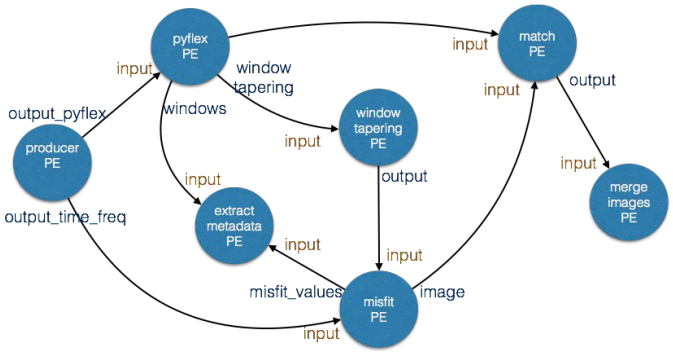


Figure 5: Misfit calculation `dispel4py` workflow.

The misfit analysis is computationally intensive and we used `dispel4py` to develop two scalable workflows that are suitable for the various platforms available in the VERCE infrastructure:

- *Phase 1 – Preprocess*: a preprocessing workflow that prepares and aligns the observed traces and the synthetic modelling results; as in the cross-correlation a series of functions is applied to the traces, in a trivially parallel computation.
- *Phase 2 – Misfit Calculation*: the workflow shown in figure 5 in which the misfit is calculated using various methods and images are created for the visual comparison of the observed data and the model.

There are several methods for calculating the misfit:

- The entire dataset of the observation and the modelling outputs are compared by calculating the misfit.
- A selection of windows is extracted from the input data before computing the misfit, using Pyflex[18], a Python port of the FLEXWIN algorithm. A taper function is then applied to the windows.

The workflow calculating the misfit of the preprocessed real and synthetic data is shown in Figure 5. At the start of the computation the producerPE emits preprocessed, aligned pairs of observed and synthetic traces produced by the preprocessing workflow. Both the window selection with subsequent misfit calculation and the direct misfit calculation are applied in the same workflow. This uses the "Tee" functionality of `dispel4py`: Connecting the pyflex PE and the misfit PE to the same output of the producer means that both consumers receive copies of the same data units. The results

[18] http://krischer.github.io/pyflex/

from both calculations are then joined in the MatchPE using group-by. Finally an image is created from the matched results that shows the windows as well as the time frequency images for each component. Also the metadata of the Pyflex and misfit calculations are available for reference along with the images.

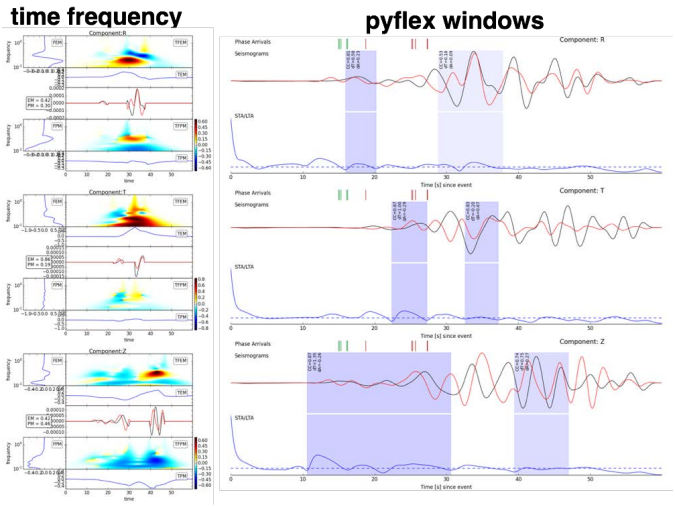


Figure 6: Misfit results visualisation.

An example image created from the outputs of the misfit calculation is shown in figure 6.

This workflow was initially tested on a local machine with a small dataset to ensure correctness. Later, it was executed on the HPC platform via the VERCE gateway.

7.3 Evaluation Platforms: DCI features

Four platforms have been used for our experiments: Terracorrelator, the Super-MUC cluster (LRZ), Amazon EC2, and the Edinburgh Data-Intensive Machine (EDIM1). These are described below and summarised in Table 2.

The Terracorrelator[19] is configured for massive data ingest in the environmental sciences at the University of Edinburgh. The machine has four nodes, each with 32 cores. Two nodes are Dell R910 servers with 4 Intel Xeon E7-4830 8 processors, each with 2TB RAM, 12TB SAS storage and 8Gbps fibre-channel to storage arrays. We used one 32-core node for our measurements.

Super-MUC[20] is a supercomputer at the Leibniz Supercomputing Centre (LRZ) in Munich, with 155,656 processor cores in 9,400 nodes. Super-MUC is based on the Intel Xeon architecture consisting of 18 Thin Node Islands and one Fat Node Island. We used 16 Thin (Sandy Bridge) Nodes, each with 16 cores and 32 GB of memory, for the measurements.

On the Amazon EC2 the Storm deployment used an 18-worker node setup. We chose Amazon's T2.medium instances[21], provisioned with 2 vCPUs and 4GB of RAM. Amazon instances are built on Intel Xeon processors operating at 2.5GHz, with Turbo up to 3.3GHz. We used 18 VMs for our measurements.

[19] http://gtr.rcuk.ac.uk/project/F8C52878-0385-42E1-820D-D0463968B3C0

[20] http://www.lrz.de/services/compute/supermuc/systemdescription/

[21] http://aws.amazon.com/ec2/instance-types/

EDIM1[22] is an Open Nebula[23] linux cloud designed for data-intensive workloads. Backend nodes use mini ITX motherboards with low powered Intel Atom processors with plenty of space for hard disks. Each VM in our cluster had 4 virtual cores – using the processor's hyperthreading mode, 3GB of RAM and 2.1TB of disk space on 3 local disks. We used 14 VMs for our evaluations.

Table 2: DCI features

Load	Terracorrelator	Super-MUC	Amazon EC2	EDIM1
DCI type	shared-memory	cluster	cloud	cloud
Enactment systems	MPI, multi	MPI, multi	MPI, Storm, multi	MPI, Storm, multi
Nodes	1	16	18	14
Cores per Node	32	16	2	4
Total Cores	32	256	36	14
Memory	2TB	32GB	4GB	3GB
Workfklows	xcorr, int_ext, sentiment	xcorr, senti-ment	xcorr	xcorr, int_ext, senti-ment

8. MONITORING FRAMEWORK

`dispel4py` supports automatic scaling in a parallel environment by creating multiple instances of a PE and distributing the data flow according to load. However, it is possible to further improve performance by optimising:

- the partitioning of the workflow graph and minimising the number of data transfers
- the number of processes that is assigned to each PE or graph partition

In our tests we have seen promising improvements by applying these two strategies.

In the current stable release of `dispel4py`, users can test optimisations manually by creating partitions and assigning processes to PEs or partitions explicitly. However, profiling of applications and the subsequent analysis of collected data is an important tool for detecting bottlenecks in an application and diagnosing issues. To address this we created a monitoring framework for `dispel4py` which collects data in the form of timestamps from a workflow during enactment.

Collecting the raw timestamp data during the enactment of a workflow allows to subsequently extract and analyse performance and provide diagnostics. The timestamps are stored in the scalable NoSQL database MongoDB[24] providing high-level query functionality.

In the future, based on the monitoring framework, we plan to explore various semi-automated and fully automated optimisation strategies. We will create a benchmark package that a user can easily deploy on a new platform to help

[22] https://www.wiki.ed.ac.uk/display/DIRC

[23] http://opennebula.org

[24] https://www.mongodb.org/

with the collection of performance data and optimisation of user-defined workflows.

The `dispel4py` monitoring service is a web service that provides access to a number of web pages that outline profiling data:

- An overview of the workflow run with information such as the mapping, the number of processes, the assignment of PEs to processes and the total runtime. It also shows a visualisation of the workflow graph.
- Details on timings of methods, such as the total times and average times for methods `process`, `read` and `write`, broken down by PE and PE instance.
- Diagrams illustrating the times spent in each method, for each PE and each process.
- A timeline of the processes and interactions.
- Details of communication times between processes.

Figure 7 shows the main page of the monitoring service, listing the job runs for which data was collected, their mapping and the start and end times. More details are linked from this page. The information may be used to identify possible bottlenecks and formulate optimisation strategies.

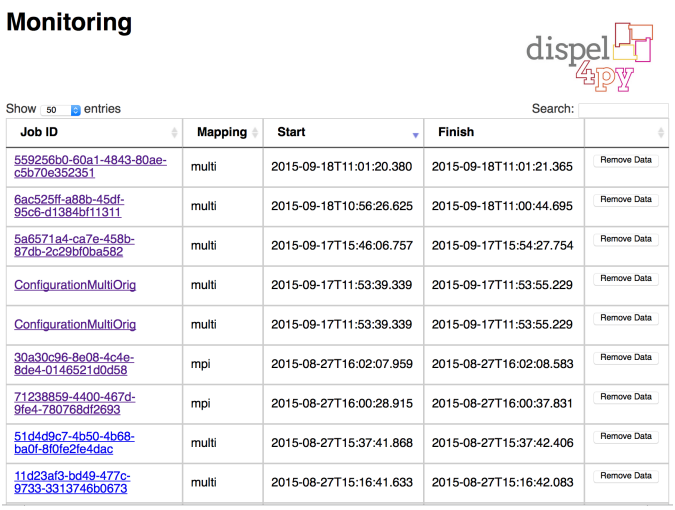


Figure 7: Monitoring service – main page

From the main page the job link takes the user to the job information page, shown in figure 8, which details more information on a job, such as the mapping, the total runtime, and the number of processes and their assignments to PEs. It also displays the visualised graph.

From here, further links point to pages showing:

- summary profiles of the job (the total processing time per PE, and times for reads and writes), shown in figure 10,
- diagrams that illustrate the processing times for each PE or process, shown in figure 9,
- a diagnostics page with details of the communication times between PEs, allowing to detect bottlenecks.

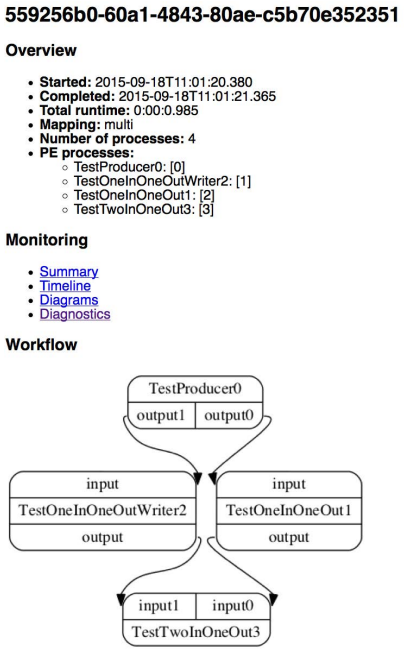


Figure 8: Monitoring service – Detailed job information

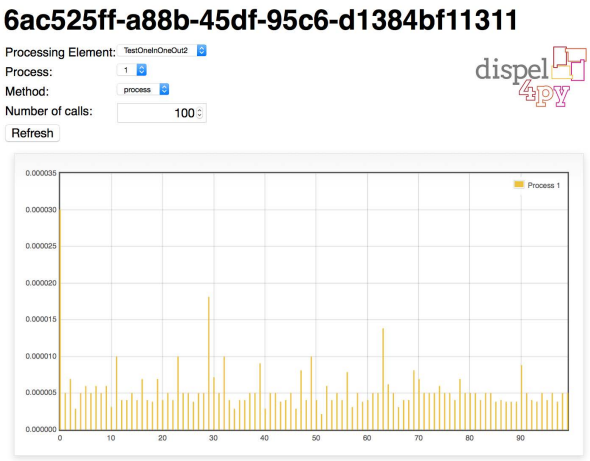
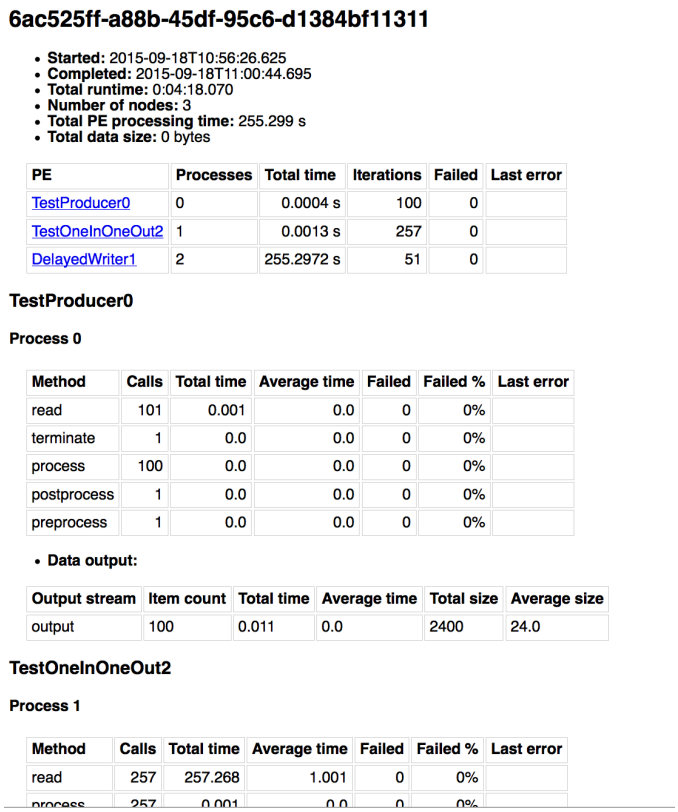


Figure 9: Monitoring service – method timing plot

Figure 9 shows an example diagram for the times for each processing iteration of one PE.

Obviously performance monitoring comes at a cost and the feature would be turned off during large-scale production runs. The monitoring framework can be easily switched on or off when executing a workflow. We envisage that a user would test a workflow on a DCI platform for a short time on a subset of the input data to collect relevant monitoring information. This data would feed into the selection of optimisation strategies for the assignment of processes to PEs and the encapsulation of PEs in partitions, resulting in optimal use of compute resources in large-scale production runs.

As a first step, an optimisation strategy can be inferred and applied manually after identifying bottlenecks. However, `dispel4py`'s design allows to plug in optimisers that



6ac525ff-a88b-45df-95c6-d1384bf11311

- **Started:** 2015-09-18T10:56:26.625
- **Completed:** 2015-09-18T11:00:44.695
- **Total runtime:** 0:04:18.070
- **Number of nodes:** 3
- **Total PE processing time:** 255.299 s
- **Total data size:** 0 bytes

PE	Processes	Total time	Iterations	Failed	Last error
TestProducer0	0	0.0004 s	100	0	
TestOneInOneOut2	1	0.0013 s	257	0	
DelayedWriter1	2	255.2972 s	51	0	

TestProducer0

Process 0

Method	Calls	Total time	Average time	Failed	Failed %	Last error
read	101	0.001	0.0	0	0%	
terminate	1	0.0	0.0	0	0%	
process	100	0.0	0.0	0	0%	
postprocess	1	0.0	0.0	0	0%	
preprocess	1	0.0	0.0	0	0%	

- **Data output:**

Output stream	Item count	Total time	Average time	Total size	Average size
output	100	0.011	0.0	2400	24.0

TestOneInOneOut2

Process 1

Method	Calls	Total time	Average time	Failed	Failed %	Last error
read	257	257.268	1.001	0	0%	
process	257	0.001	0.0	0	0%	

Figure 10: Monitoring service – Profiling information

modify and annotate workflow graphs automatically prior to handing them over to the enactment engine. Alternatively the user could be presented with a number of options in an interactive session and select according to their experience with a particular platform or workflow.

In the future, enactment platforms may also enable a user to pause and restart a workflow during execution, allowing to rebalance PEs and partitions according to profiling information.

9. CONCLUSIONS AND FUTURE WORK

In this paper we presented `dispel4py`, a novel Python library for streaming, data-intensive processing. The novelty of `dispel4py` is that it allows its users to express their computational need as a fine-grained abstract workflow, taking care of the underlying mappings to suitable resources. Scientists can use it to develop their scientific methods and applications on their laptop and then run them at scale on a wide range of e-Infrastructures without making changes.

We demonstrate with a realistic scenarios borrowed from the field of seismology that `dispel4py` can be used to design and formalise scientific methods. `dispel4py` is easy to use, it requires very few lines of Python code to define a workflow, while the PE functionality can be re-used in a well-defined and modular way by different users, in different workflows and executed on different platforms via different mappings.

In the near future we envisage the addition of optimisation mechanisms based on a number of features, such as a monitoring framework and diagnostic tools that can select the best target DCIs and enactment modes automatically. A type system would also be an interesting addition to prevent users from experiencing runtime errors if the types produced by a PE do not match the input types of the consumer PE. Additionally, we aim to explore more mappings, such as improving the existing prototype for *Apache Spark*.

Acknowledgment

This research was supported by the VERCE project (EU FP7 RI 283543) and the Terracorrelator project (funded by NERC NE/L012979/1).

10. REFERENCES

[1] M. Atkinson, M. C. S. Claus, R. Filgueira, and et al. Verce delivers a productive e-science environment for seismology research. In *IEEE International eScience Conference*, 2015.

[2] M. P. Atkinson, C. S. Liew, M. Galea, P. Martin, A. Krause, A. Mouat, Ó. Corcho, and D. Snelling. Data-intensive architecture for scientific knowledge discovery. *Distributed and Parallel Databases*, 30(5-6):307–324, 2012.

[3] R. Filguiera, I. Klampanos, A. Krause, M. David, A. Moreno, and M. Atkinson. Dispel4py: A python framework for data-intensive scientific computing. In *Proceedings of the 2014 International Workshop on Data Intensive Scalable Computing Systems*, DISCS '14, pages 9–16, Piscataway, NJ, USA, 2014. IEEE Press.

[4] MPI Forum. MPI: A message-passing interface standard. *IJ of Supercomputer Applications*, 8:165–414, 1994.

APPENDIX

A. DISPEL4PY MISFIT WORKFLOW

The listing below constructs the graph of the misfit workflow as described in section 7.2. The resulting `dispel4py` graph is illustrated in figure 5. Implementation details of the processing elements are omitted.

```
graph = WorkflowGraph()

producer_PE = StreamProducer()
pyflex_PE = PyflexPE()
misfit_PE = MisfitPE()
match_PE = MatchComponents()
extract_metadata_PE = ExtractMetadataPE()
window_tapering_PE = WindowTaperingPE()
merge_images_PE = MergeImagesPE()

graph.connect(producer_PE, "output_pyflex",
              pyflex_PE, "input")
graph.connect(producer_PE,
              "output_time_frequency",
              misfit_PE, "input")
graph.connect(pyflex_PE, "image",
              match_PE, "input")
graph.connect(pyflex_PE, "window_tapering",
              window_tapering_PE, "input")
graph.connect(window_tapering_PE, "output",
              misfit_PE, "input")
graph.connect(misfit_PE, "image",
              match_PE, "input")
graph.connect(match_PE, "output",
              merge_images_PE, "input")
```